6 Performance Characterization: Asymptotic Analysis

In this chapter, we will look at empirical and analytical ways of understanding the performance of a piece of code. Besides correctness, performance is an important consideration in designing and choosing an algorithm. In quantifying performance, users generally wish to know whether to choose this design or its rivaling alternative—or whether the present design will be fast enough for the job and how well it can scale with larger inputs.

Two main kinds of resources are of particular interest: time and space. This chapter focuses mostly on time. We will begin by looking at how to experimentally study the running time and discussing their limitations. Following that, we devote the rest of the chapter to quantifying performance analytically. To this end, we discuss asymptotic notations and their use in describing running time. We then extend our analysis technique to recursive programs by discussing recurrence relations.

Performance. How fast is the code we just wrote? If it can be easily run, we can measure the time required to work on a particular input. We can then run the code on a wide variety of inputs and extrapolate the behavior. However, this may or may not be representative of how the code actually behaves in general. But what if we do not have an implementation? This is where analytical methods come in, allowing for discussing the performance of different ideas and concrete implementations alike.

6.1 Experimental Study of Running Time

With an implementation of an algorithm, we can study its running time behavior by running it and recording how long it takes. It is best to try the code out on a wide variety of inputs. By recording the start time and the finish time after running the code, we can determine the time used, which is simply the difference between these time points. In Java, there are two commonly used functions to determine the time:

- `System.nanoTime()` returns the current time, with respect to a reference point, in nanoseconds (ns).
- `System.currentTimeMillis()` returns the current time in milliseconds.

Because of the greater precision, `nanoTime` is generally the preferred choice, hence writing, for example:

```
long startTime = System.nanoTime();
// run code that is going to be measured
long endTime = System.nanoTime();
long timeUsed = endTime - startTime; // in ns
```

Wall-Clock Time and Interpretation

The time measured in this way is known as a *wall clock* time because it is the time that has elapsed in real life while the program does the work. This gives a reasonable sense of a program's performance, though we could see interference from other things running on the same machine or other factors.

Often, we study wall-clock time performance because we want to predict the behavior of such a program in the future, on input that we have not seen before. Therefore, we are interested in how the running time varies as the size and the structure of the input changes. To do this, just like in science experiments, we perform independent runs on many test inputs of various sizes and configurations. Plotting it on a time vs. input size scale (like below) can provide us with a good sense of the program's overall behavior.

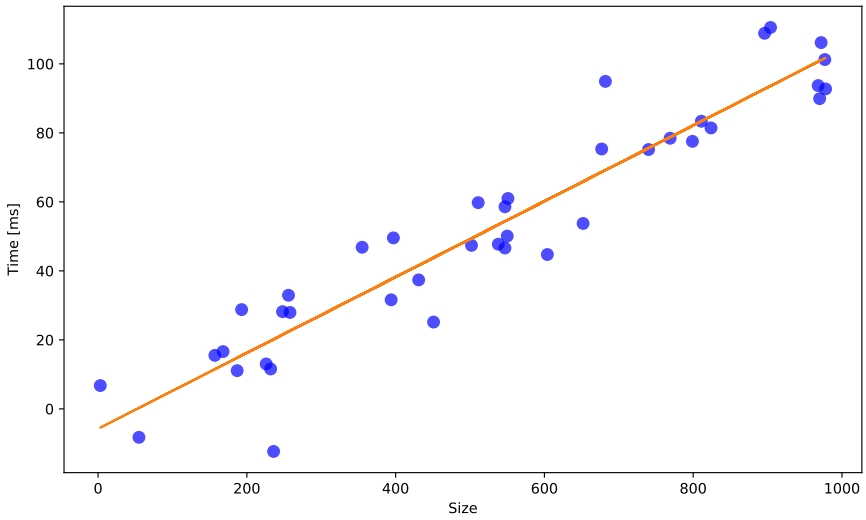


Figure 6.1: Example wall-clock time data (dots) and its best-fit line.

When we take a look at this plot, most of us will agree that it exhibits a linear trend—as in, the best-fit line scales linearly with the input size. Indeed, the best-fit line, obtained via linear regression, confirms this trend.

Aside from linear, what other common trends are there? Below, we review some functions that commonly arise in describing performance trends.

6.2 Common Functions for Performance Characterization

There are seven basic functions that will keep coming up when we discuss performance trends. In this section, we will review them briefly.

The Constant Function

The constant function is the simplest function, $f(n) = c$ for some *fixed* constant c. For example, $c = 13, c = 17, c = 10^{10}$. What happens here is, regardless

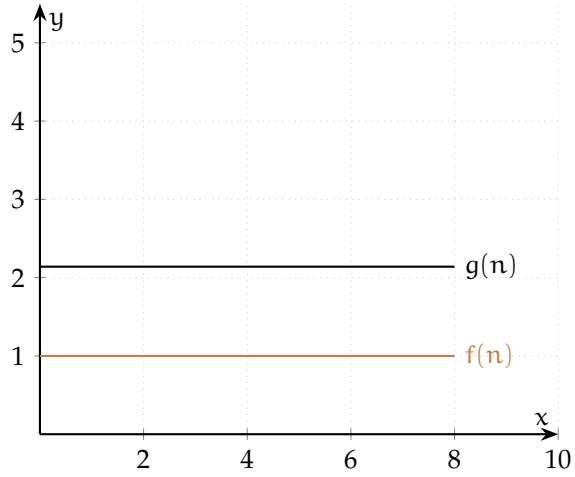


Figure 6.2: Constant functions $f(n) = 1$ and $g(n) = 2.14$.

of the input size n, the function remains at that constant c. The constant function is useful in algorithm analysis as it captures the running time of basic/primitive operations as well as a few other cheap operations. If we plot a constant function, the curve looks flat (see examples in Figure 6.2).

The Linear Function

This is the function $f(n) = c \cdot n + b$ for some constants c and b. Given an input of size n, this function registers the value cn. Notice that if $f(t) = t$, then $f(2t) = 2t$—or in words, the value given by such a function doubles when the input doubles. Interestingly, this comes up quite often in characterizing the running time of a program. For example, if the cost of touching each one element is constant, say c, then the time to go over a sequence of n elements is cn. In picture, the curve for such a function looks as follows.

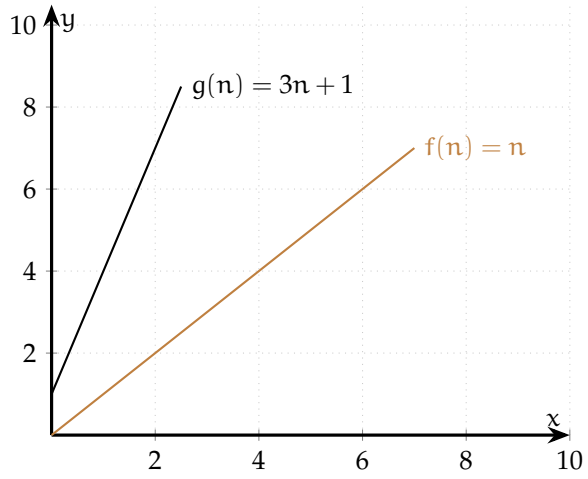


Figure 6.3: Linear functions.

The Logarithmic Function

This is the function $f(n) = \log_b n$ for some constant $b > 1$, where the meaning of the log function is given by

$$x = \log_b n \iff b^x = n.$$

In computer science, the most common base is 2, so we tend to write $\log n$ to mean $\log_2 n$. For practical purposes of algorithm analysis, $\log n$ (or $\log_2 n$) is the answer to the question "how many times can we divide by 2 before n goes to or below 1?" Graphically, this is what it looks like:

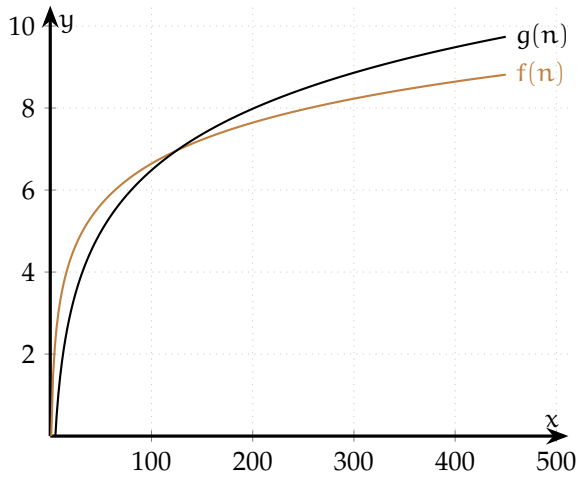


Figure 6.4: Logarithmic functions $f(n) = \log_2(n)$ and $g(n) = 1.5\log_2(n/5)$.

At this point, we should remind ourselves of some logarithmic identifies via examples:

$$\begin{aligned}
\log(4n) &= \log 4 + \log n = 2 + \log n \\
\log(n/2) &= \log n - \log 2 = \log n - 1 \\
\log(n^2) &= 2\log n; \log(n^{100}) = 100\log n \\
\log(2^n) &= n \\
2^{\log n} &= n
\end{aligned}$$

The N-Log-N Function

That is, $f(n) = n \log n$. There is probably no good reason why this function should occur in nature, but it does come up often as the running time of many programs. Figure 6.5 shows a plot for both a linear function and a nearly-identical n-log-n function.

The Quadratic Function

The quadratic function is $f(n) = n^2$ (read: n squared). It appears frequently in programs with nested loops, for example, when there are two loops

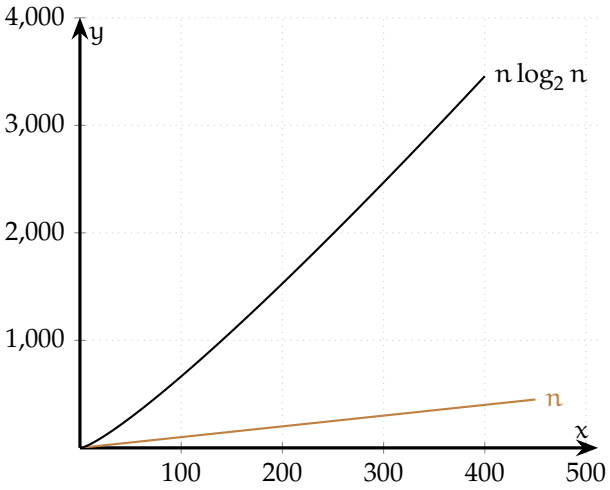


Figure 6.5: Between the function n and a nearly-identical n-log-n variant.

```
for i = 0, 1, 2, ..., n - 1:
  for j in 0, 1, 2, ..., n - 1:
    // do some constant work
```

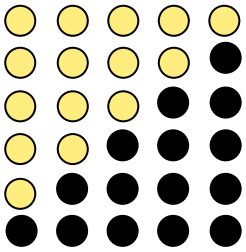
```
for i in 0, 1, 2, ... , n - 1:
  for j in 0, 1, ..., i - 1:
    // do some constant work
```

The code on the left clearly does n iterations, each containing n iterations of constant-time work. Hence, this takes a total of $n \times n \times c = c \cdot n^2$ time, where c is how long it takes to carry out the constant-time work. For the code on the right, in the first iteration, the inner loop does 1 unit of work. Then, 2 units, then 3 units, giving a total work of

$$(1 + 2 + ... + n)c = \frac{n(n+1)c}{2},$$

which follows from the standard summation formula.

Summation. The sum $\Delta = 1 + 2 + 3 + \cdots + n$ can be found by arranging two copies of Δ dots as illustrated with $n = 5$. Clearly, $2\Delta = n(n+1)$.



The Cubic Function

This is $f(n) = n^3$, which sometimes occurs. As the last of polynomially-related functions, how big is the cubic function for large n compared to previously-discussed functions? As it turns out, for large n, the higher the exponent, the biggest the function value. Hence, we have that

$$\text{const} < \log n < n < n \log n < n^2 < n^3.$$

The Exponential Function

This is $f(n) = b^n$ for a positive constant b (called the base). The readers should review the exponent rules. Related to this, here is one identity that will come up again and again.

Geometric Sum. The geometric sum formula for powers of 2 states that

$$1+2+4+\cdots+2^n=\sum_{k=0}^{n}2^k=2^{n+1}-1$$

There are many ways to prove this fact. For now, let us prove it informally using bit tricks. First, consider the following binary number N:

$$N=(\underbrace{111111...1}_{n\text{ of }1's})_2$$

From binary addition, we know that $N+1$ is equal to the following:

$$\begin{array}{cccccccc} 1 & 1 & 1 & 1 & \dots & 1 & 1+ \\ & & & & & & 1 & , \\ \hline 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 \end{array}$$

which is to say that $N+1=2^{n+1}$. But what is the numerical value of N itself? By the very definition of base-2 numbers, we have

$$N=2^0+2^1+2^2+\dots 2^n=\sum_{k=0}^{n}2^k$$

Altogether, we have that $N+1=2^{n+1}$, so then $N=2^{n+1}-1$. That is,

$$\sum_{k=0}^{n}2^k=2^{n+1}-1.$$

6.3 Beyond Experimental Analysis

Experimental analysis looks fine and dandy, especially when we have a properly implemented piece of software that needs fine-tuning. But there are a few major limitations to always conducting experiments:

- Experiments can only be done on a limited set of test inputs and may not reflect the behavior of all possible inputs.
- We need an implementation to run the experiment with. (This can be costly especially if all we want is to compare options before we implement a solution.)

To move beyond empirical analysis, we want an approach that

- can evaluate the relative efficiency of algorithms—at least approximately—that are independent of the hardware/software platforms.
- can be applied by looking at just the high-level description, without the need to fully implement it.
- can take into account all possible inputs

How can we satisfy all these requirements? One solution is to count the number of primitive operations. These are operations that can be done simply, in a constant amount of execution time—and indeed, we can time them and see that they run in constant amount of time. Examples include:

- Assigning an identifier to an object
- Performing arithmetic operations on normal-sized numbers
- Accessing an element in a fixed-size array by index
- Returning from/calling a function (excluding the work performed within the function)

Quite amazingly, on modern computer systems, the cost of primitive operations only vary a little (by constant factors) across hardware/software platforms—if we define the primitive operations well.

Therefore, we could, in theory, come up with a function

$$f(I) = \text{the number of primitive operations performed on input } I.$$

But that does not satisfy all of our needs and it is rather cumbersome to talk about. We will look for a simpler alternative.

Function of Input Size

Instead of discussing the running time of each individual input, we settle to use $f(n)$ to characterize the number of primitive operations performed when the input has size n. This means regardless of the actual input, as long as the input has size n, we want it to behave more or less like $f(n)$. But then, two questions arise:

1. *How do we measure the input's size?* Usually, there is a natural measure for the problem. For example, if the input is a sequence, we can use the number of elements in that sequence.
2. *How do we handle the fact that an algorithm can take drastically different amounts of time on different inputs even of the same size?* For starters, we aim the highest and use the worst-case behavior. This is because average-case behaviors tend to be difficult to pinpoint, and if we use the worst-case behavior, optimizing for it means it will work well no matter what. Hence, we are optimizing for a stronger requirement.

6.4 Asymptotic Analysis

In our discussion thus far, we wanted to count the number of (primitive) operations an algorithm performs on the worst possible (i.e., most difficult) input of a certain size n. In later discussions, we will consider the behaviors of other kinds of inputs as well.

A few problems need to be addressed: First, we are lazy—finding the exact count is a nonstarter. Second, the exact count is generally useless because when mapped down to the machine level, specifics of the platform matter more than slight differences in constants. Third, we want something clean to talk about. Hence, our goal is to find a first-order approximation that has good predictive power. This brings us to a family of asymptotic notation.

The Big-O Notation

Definition 6.1 (Big-O)**.** Let $f(n)$ and $g(n)$ be functions mapping positive reals to positive real numbers (i.e., $f, g : \mathbb{R}_+ \to \mathbb{R}_+$). Then, $f(n)$ is $O(g(n))$ **(read:** f is *big-O* of g**)** if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

To say $f(n)$ is $O(g(n))$, we often write $f(n) \in O(g(n))$ and $f(n) = O(g(n))$. For example, $f(n) \in O(n^2)$ is read "$f(n)$ is big-O of n^2," and $h(n) = O(n)$ is read "$h(n)$ is big-O of n."

While this definition is usually convenient to work with, it does not give us much intuition. The following alternative gives more clue: We say $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geqslant 1$ such that $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

The formal definition can be scary, but the gist of it is simple: For large n (larger than a threshold), $f(n)$ is at most another function $g(n)$ up to a constant factor. That is to say, for smaller values of $n < n_0$, $f(n)$ and $g(n)$ may be wild and have no relationship, but for large values of $n \geqslant n_0$, $f(n)$ is upper-bounded by a constant times $g(n)$. Ignoring constants, $f(n) = O(g(n))$ says $f(n)$ is at most $g(n)$ for large enough inputs.

We will now try to develop an intuitive understanding of the notation.

(i) The Big-O of a polynomial function function is its dominant term.

> **Example.** Consider several examples below. For each of them, we can verify it using the limit definition.
>
> - $10n^7 + n^2 + 900n \log n + 10^{100} \in O(n^7)$ as one can verify
>
> $$\lim_{n\to\infty} \frac{10n^7 + n^2 + 900n \log n + 10^{100}}{n^7} = 10 < \infty.$$
>
> - $n^2 + n \log n \in O(n^2)$
> - $2n + n \log n \in O(n \log n)$

(ii) Big-O adds and multiplies in natural ways:

> **Example.** If we perform n tasks, each taking $O(n)$ time, the total time spent is $O(n^2)$.

> **Example.** Say we perform *two* tasks: the first task takes $O(n^2)$ time and the second task takes $O(n \log n)$ time. Altogether, the time we spend equals the sum of the two tasks. It should be $O(n^2 + n \log n) = O(n^2)$ because that term dominates.

Let us formalize this slightly more in the following lemma, whose proof appears as Exercise 6.7.

Lemma 6.2. Let $d(n) = O(f(n))$ and $e(n) = O(g(n))$. Then,

(i) $d(n) + e(n) = O(f(n) + g(n))$, and

(ii) $d(n) \cdot e(n) = O(f(n) \cdot g(n))$.

A Fashion Issue. While the definition gives us some leeway, do use the tightest and simplest term to characterize functions. As an example, by definition, $f(n) = 10n^3 + 2$ is $O(n^3)$, even $O(n^4)$; however, it is customary to write $O(n^3)$ because it is the simplest and closest expression.

Big-Oh is like this: How long did it take you to drive to school this morning? The exact answer: 1 hour, 3 minutes, 2 seconds, 15 milliseconds, and 5 nanoseconds. Big-Oh gives us a way to say "no more than 2 hours" ignoring the nitty-gritty detail. Now while it's also correct to say "no more than 12 hours." We would rather say "2" than "12" because it is a better approximation and as succinct.

The Big-Theta Notation

To say $f(n)$ is $\Theta(g(n))$, we often write $f(n) \in \Theta(g(n))$ and $f(n) = \Theta(g(n))$.

In a strict sense, big-O says one function is "upper-bounded" by another function. Although it is fashionable to give a tight upper bound, tightness is not a requirement. Mathematically, there is a related notion that precisely says a function f behaves in essentially the same way as g.

Definition 6.3 (Big Theta). $f(n)$ is $\Theta(g(n))$ (**read:** f is *theta* of g) if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

for some constant $c > 0$.

Alternatively, we say that $f(n)$ is $\Theta(g(n))$ if it is true that $f(n) = O(g(n))$ and $g(n) = O(f(n))$. Hence, extending our understanding of Big-O, we have that $f(n)$ is upper-bounded by $g(n)$, up to a constant, and $g(n)$ is also upper-bounded by $f(n)$, up to a constant. Hence, $f(n)$ and $g(n)$ are about the same, up to constants.

Example. $f(n) = 10n^3 + 11n^2 + 3$ is $\Theta(n^3)$ because

$$\lim_{n\to\infty} \frac{f(n)}{n^3} = 10.$$

Notice, however, that although $f(n) = O(n^4)$, $f(n)$ is not $\Theta(n^4)$ as we can see that

$$\lim_{n\to\infty} \frac{f(n)}{n^4} = 0 \not> 0.$$

As the following lemma states, big theta expressions add and multiply just like Big-O. The proof of this is left as an exercise to the reader in Exercise 6.8.

Lemma 6.4. Let $d(n) = \Theta(f(n))$ and $e(n) = \Theta(g(n))$. Then,

(i) $d(n) + e(n) = \Theta(f(n) + g(n))$, and

(ii) $d(n) \cdot e(n) = \Theta(f(n) \cdot g(n))$.

Note on Formality

Technically speaking, the big-O and big-Θ of a function, as in $O(n)$, is a set. In particular, for any function g,

$$O(g(n)) = \left\{ f(n) \;\middle|\; \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \right\}$$

and

$$\Theta(g(n)) = \left\{ f(n) \;\middle|\; \lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \right\}.$$

Other than $O(\cdot)$ and $\Theta(\cdot)$, we say that $f(n)$ is $\Omega(g(n))$, written $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$, if $g(n) = O(f(n))$.

Therefore, it is more proper to say $f(n) \in O(g(n))$, but most people do not do that. We tend to write $f(n) = O(g(n))$. Abusing notation, we also use Big-O in expressions such as $n + O(\log n)$. The presence of the Big-O in an expression such as $h(n) + O(g(n))$ simply means that expression is at most $h(n) + k \cdot g(n)$ for some constant $k > 0$. Hence, the expression $n + O(\log n)$ just means $n + k \log n$ for some $k > 0$.

6.5 Running Time Analysis

Having developed the machinery for talking about how fast a piece of code runs, we will now talk about how to analyze such a program to know how fast it runs.

As a warm-up, consider the following simple program:

```
int moo(int x) { return x + 2; }
```

We see that if we feed this function any number x, the only thing it will do is to compute $x + 2$ and return that value. This requires a few (still constant) number of primitive operations. So we can make the following claim:

> On input x a number, `moo` takes at most c time for some constant c—or in Big-O notation $O(1)$ time.

To justify this, we argue that both computing $x + 2$ and returning that value can be done in a constant number of primitive operations.

As a second example, consider the following code, where each line is annotated with how long it takes to run.

```
1 int foo(int[] lst) {
2     int theMax = lst[0];   // constant time - primitive ops
3     for (int i=1;i<lst.length;++i) {  // rep n - 1 times
4         if (theMax < lst[i])        // compare (prim op)
5             theMax = lst[i];        // assignment (prim op)
6     }
7     return theMax           // just a return
8 }
```

To count steps, we can break down the time into:

- work done on Line 2, which is constant $O(1)$;
- work done on Lines 3–5, which we will analyze; and

- work done on the "return" line, which is constant $O(1)$

Hence, the total time will be $O(1)$ steps plus however long the work done on Lines 3–5 takes. This brings us to the question: how costly are they?

The new challenge here is the presence of a loop. For notational convenience, denote by n the length of `lst`. Now as we know, a loop just says repeat for each member inside it aside from the element at index 0. We know that the inside of this loop does the same amount of work (more or less) in every iteration. In a more elaborate form, we have

Iteration Count	Steps Taken
$i = 1$	some constant c
$i = 2$	some constant c
$\vdots$	
$i = n - 1$	some constant c

Therefore, Lines 3 – 5 take $c(n-1)$ steps in all, which is $O(n)$. Overall, this is an $O(n)$ algorithm.

Naïve Sequence Uniqueness

For a more involved example, we will consider a sequence uniqueness problem. In this problem, we want to find out whether all elements of a given array are distinct from each other. That is, we will write a function that takes an array and return true if there are no duplicates in the input (and false otherwise). For example:

- `[1, 3, 2, 5, 1]` is *not* unique. The number `1` is repeated twice.
- `[1, 4, 3, 5, 7]` is unique.

We can imagine writing the following piece of code: try all possible pairs of numbers and see if they have the same value, as illustrated by the following pseudocode:

```
def is_unique(lst):
    n = len(lst) # the length of lst
    for i in 0..(n-1):
        for j in (i+1)..(n-1):
            if lst[i] == lst[j]:
                return False
    return True
```

It is worth noting that for each value of `i`, the value of `j` runs from `i+1` through `n-1`. This means, for example, when `i` is 0, `j` ranges from `1` to `n-1` (inclusive).

To analyze the running time of this code, we will quantify the time complexity by counting the number of primitive operations. But to begin, for any given `i` and `j`, what is the running time of the following two lines?

```
if lst[i] == lst[j]:
    return False
```

The actual number of operations will depend on the outcome of the equality test, but in any case, these two lines run using at most a constant number of primitive operations, so $O(1)$ time—or some constant k.

As we proceed, let $n = \texttt{len(seq)}$ for notational convenience. We will now tackle the time required by the nested for-loop portion.

First, notice that the program may exit early but in the worst case, the outer loop goes over indicies $i = 0, 1, 2, \ldots, n-1$. Therefore, if $f(i)$ is the time taken within the loop with that value of i. The total time of the nested for-loop portion is the summation

$$f(0) + f(1) + f(2) + \cdots + f(n-1) = \sum_{i=0}^{n-1} f(i).$$

We just need to know each $f(i)$. Now notice that for each i, the inner loop goes over $j = i+1, i+2, ..., n-1$. This means that for each i, there are $n-1-i$ values of j. Further, the inside of the inner loop requires at most k time, as discussed previously. For a value of i, because it takes at most k time for each j and there are $n-1-i$ values of j, we know that

$$f(i) = k \times (n-1-i).$$

Taken together, the nested for-loops require a total of

$$f(0) + f(1) + f(2) + \cdots + f(n-1) = \sum_{i=0}^{n-1} k \times (n-1-i) = k \sum_{i=0}^{n-1} (n-1-i)$$

The summation $(n-1) + (n-2) + \cdots + (n-1-(n-1))$ is identical to $1 + 2 + 3 + \cdots + (n-1)$, which is equal to $n(n-1)/2$ by the summation formula.

Hence, this portion alone therefore runs in $O(n^2)$. But this code contains other constant-time statements, so the grand total running time is $O(n^2 + c) = O(n^2)$. We conclude that the function `is_unique` runs in $O(n^2)$ time.

Simple Summation

As another example, consider the following Java code that finds the sum of a list of numbers:

```java
int sum(List<Integer> numbers) {
    int total = 0, n = numbers.size();
    for (int i=0;i<n;i++)
        total += numbers.get(i);
    return total;
}
```

Skipping the trivial portions of the code, we will pay attention to the for-loop that goes over each element of the list. Since we have no idea what kind of list this is, it is possible that `.get` will take different amounts of time depending on which list implementation it is.

Scenario I: Suppose the input list is, for example, an `ArrayList`, and according to the documentation, each `.get(i)` takes constant time. Then, we have

that because the loop runs for n iterations and each iteration takes $O(1)$ time, it comes out to $O(n)$ time total. The other miscellaneous steps take at most constant time. We conclude that the `sum` function runs in $O(n)$ time, where n is the length of the input list.

Scenario II: Suppose the input list is, for example, a `LinkedList`, and according to the documentation, each `.get(i)` takes time proportional to i—i.e., $c \cdot i$ for some constant $c > 0$. Then, the loop runs $i = 0, 1, 2, \ldots, n-1$, so the running time of this loop is

$$\sum_{i=0}^{n-1} c \cdot i = \frac{n(n-1)}{2} = O(n^2)$$

Again, the other miscellaneous steps take at most constant time. In this case, we conclude that the `sum` function runs in $O(n^2)$ time, where n is the length of the input list.

A Mysterious Function

We will look at a function, without attempting to explain what it is doing, and will try to understand its performance characteristics.

```
1 int[] baz(int array[]) {
2     int[] ans = new int[array.length];
3     for (int i=0;i<array.length;i++) {
4         int[] copy = Arrays.copyOfRange(array, 0, i+1);
5         Arrays.sort(copy);
6         ans[i]=copy[i];
7     }
8     return ans;
9 }
```

We will now analyze the running time of the `baz` function. Unlike any of the previous examples, this makes a number of function calls.

Let n be the length of the input array (i.e., n is `array.length`). Several things are clear immediately. First, the line

```
int[] ans = new int[array.length];
```

allocates a new array of length n. This costs us $O(n)$ as Java has to find space big enough to fit these `int`s and after that, it fills the whole array with `0`s. It loops through an write `0`s for a total of n times. Other than that, we know the statement `return ans;` takes constant time.

If anything, the bulk of the cost comes from the for-loop, and it is not clear a priori how to analyze it. Some questions we need to answer:

- How long does it take per iteration? At a glance, this seems to vary across iteration.
- To answer the previous question, we need to know—How long is each of these function calls `Arrays.copyOfRange` and `Arrays.sort`?

copyOfRange. If we were to write this ourselves, we would allocate a new array of right length and loop through to copy exactly these many elements.

Let us work out the cost of the functions that we call first. By looking up the documentation, we find that

- On input an array a of length m, `Arrays.sort(a)` takes time $O(m \log m)$.
- The call `Arrays.copyOfRange(a, from, until)` takes time $O(z)$ where $z = \texttt{until} - \texttt{from}$.

Next we will work out the running time of each iteration `i`. According to what we found, the cost of iteration `i` can be broken down into the following:

- the call to `copyOfRange`, which takes $O(i)$ time
- the call to `sort`, made on an array of size `i`, so it takes $O(i \log i)$ time.
- finally, the instruction `ans[i] = copy[i]` takes $O(1)$ time.

That is, iteration `i` costs us $O(i) + O(i \log i) + O(1) = O(i \log i)$—more formally, $c \cdot i \log i$ for some constant c. With this analysis, we can make a familiar table of the time taken in each iteration:

Iteration Count	Steps Taken
$i = 0$	0
$i = 1$	$c \cdot 1 \log 1$
$i = 2$	$c \cdot 2 \log 2$
$\vdots$	
$i = n - 1$	$c \cdot (n-1 \log(n-1))$

This means, in all, the total running time is

$$c \cdot 1 \log 1 + c \cdot 2 \log 2 + \cdots + c \cdot (n-1) \log(n-1) = \sum_{i=0}^{n-1} c \cdot i \log i$$

But then every $\log i$ is at most $\log n$ because $i \leqslant n - 1$. Hence, the total running time is *at most*

$$\sum_{i=0}^{n-1} c \cdot i \log n = (c \cdot \log n) \times (1 + 2 + \cdots + n - 1) = O(n^2 \log n).$$

Example: Maximum Contiguous Subsequence Sum (mcss)

We will now turn our attention to a longer example where we design an algorithm and analyze its performance. Given a sequence of possibly-negative numbers $a_1, a_2, \ldots, a_n$, find the contiguous subsequence with the greatest sum. More mathematically, we wish to

$$\text{Find } 1 \leqslant i \leqslant j \leqslant n \text{ to maximize } \sum_{k=i}^{j} a_k.$$

But what do we mean by a contiguous subsequence? Say we have as input `3 2 5 7 8 9 10 12`. Any sublist of that list is a subsequence, but it has to be contiguous, i.e., we cannot skip a number. For example, `3 2 5` is okay, but we cannot do 3 skipping 2, then 5, 7, 8.

Example. If the numbers are all positive, say `1 2 3 4 5 6 7 8`, then the more, the merrier. We might as well choose the whole sequence. The story is different when the input sequence contains negative numbers. As an example, consider the sequence `-2 11 -4 13 -5 -2`. The solution is now less obvious, but the answer is 20 (i.e., 11, −4, 13).

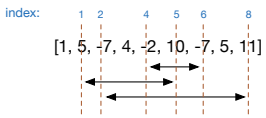
We want to write a function `mcss(a)` that takes a sequence a and returns the sum of the contiguous subsequence that has the greatest sum.

We will begin by developing a simple algorithm for solving mcss. Let us start by examining a seemingly unrelated problem that we already know how to solve: find the maximum value from a list of values. If you recall, one can write code that looks something like:

```
maxSoFar = -infty (i.e. something so tiny)
for each value in the list:
    if this value > maxSoFar:
        set maxSoFar to this value.
report maxSoFar
```

This is how we solve the max-element problem by trying all possibilities. As it turns out, we can do the same for mcss.

Our first goal is to *enumerate all possible subsequences*, so that we can compute the sum for each of them and find the maximum sum. This brings us to the second goal: *compute the sum for any sequence*.



For the first goal, a moment's thought reveals that we can describe any contiguous subsequence by

- the index of the starting point; and
- the index of the ending point

Therefore, we could enumerate all subsequences as follows:

```
for (int start=0;start<n;start++) {
    for (int stop=start;stop<n;stop++) {
        // do what you may with start..stop
    }
}
```

Equally simple is the second goal: write a simple function that computes the sum of a given subsequence:

```
int sumSubseq(int[] a, int from, int until) {
    int sum=0;
    for (int i=from;i<until;i++)
        sum += a[i];
    return sum;
}
```

Once we have these two pieces of the puzzle, it is not hard to combine them to solve the mcss problem. Code 6.1 shows the two ideas pieced together.

What's the running time of this algorithm? Although this looks like a normal two nested-loop program, there is a benign-looking line that does quite a bit of work:

Code 6.1: Brute-force minimum contiguous subsequence sum (mcss).

```
int mcss(int[] a) {
    int maxSum = Integer.MIN_VALUE;
    for (int i=0;i<n;i++) {
        for (int j=i;j<n;j++) {
            thisSum = sumSubseq(a, i, j+1);
            if (thisSum > maxSum)
                maxSum = thisSum;
        }
    }
    return maxSum
}
```

`thisSum = sumSubseq(a, i, j+1)`

How long does `sumSubseq(a, i, j+1)` take?. Take a moment to analyze this function: The function loops over $i =$ `from`, ..., `until - 1`, and per iteration, it does constant work. This is a total of `until` $-$ `from` iterations. Setting up sum = 0 and return take constant work. The total cost is therefore proportion to `until` $-$ `from` or $O(m)$, where $m =$ `until` $-$ `from`.

Now we wish to proceed like before, first deriving the cost for each i and summing them up for all i's. Let n denote the length of a. To analyze the cost for $i = 0$, we'll focus on the j loop. Several things are clear:

- j ranges from 0 to $n-1$, inclusive.
- For each j, the amount of work performed is about $j - i + 1 = j + 1$, then a constant amount from the if clause.
- Thus, the number of steps for $i = 0$ is

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}.$$

by the summation formula.

For $i = 1$, j goes from 1 to $n-1$ and for each j, the work done is $j - 1 + 1 = j$, so it is $1 + 2 + 3 + \cdots + (n-1) = (n-1)(n-2)/2$.

Following this pattern, for a general i, the value of j ranges from i to $n-1$. For each j, the amount of work performed is about $j - i + 1$, plus a constant. That is, for this i, the time required is $1 + 2 + 3 + \cdots + (n-i) = (n-i)(n-i+1)/2$.

Hence, we can analyze the total time as

$$\begin{array}{lrcl}
\text{Time when } i = 0: & n(n+1)/2 & \leqslant & n^2 \\
\text{Time when } i = 1: & (n-1)(n)/2 & \leqslant & (n-1)^2 \\
\text{Time when } i = 2: & (n-2)(n-1)/2 & \leqslant & (n-2)^2 \\
\vdots & \vdots & & \vdots \\
\text{Time when } i = n-2: & (n-[n-2])(n-[n-3])/2 & \leqslant & 2^2 \\
\text{Time when } i = n-1: & (n-[n-1])(n-[n-2])/2 & \leqslant & 1^2
\end{array} +$$

for a total of

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1),$$

which is $O(n^3)$. The last step used the sum of squares formula.

Room for Improvement. Do we see any room for improvements? We wanted a faster algorithm. To do this, we need some more thoughts. See Exercise 6.9. for an $O(n^2)$-time algorithm. In fact, we can come up with an $O(n)$-time algorithm. There indeed is a simple algorithm that runs in $O(n)$ that is pretty intuitive.

6.6 Running Time of Recursive Algorithms

How can we determine the running time of a recursive algorithm? So far, we have talked about counting the number of steps taken by an algorithm and use that as a prediction for the algorithm's running time. However, when an algorithm is recursive, it can be daunting to attempt to count the number of steps directly. For this, we will use a tool called recurrence relations.

We probably already have encountered some recurrence relations. For example, the Fibonacci recurrence $F(0) = 1, F(1) = 1, F(n) = F(n-1) + F(n-2)$; or $S(n) = S(n-1) + n$. These are simply recursive functions: functions that call themselves.

The main idea in using recurrence relations to analyze the running time of a recursive algorithm is as follows:

> Given a function that we want to analyze, we will write another recursive function—parallel to the function being analyzed but just simpler—to determine the running time.

Example: Raising A Number To A Positive Integral Power

We wish to implement a function $\texttt{pow}(b, w)$ which takes as input a nonzero real number $b \neq 0$ and a nonnegative integer $w \geqslant 0$. The function is to output the number b^w. Here is a simple recursive program we would perhaps write:

```
int pow(int b, int w) {
    if (w==0) return 1;
    else return pow(b, w-1)*b;
}
```

Let us focus on the running time of this program. The function takes two parameters as input, but we will see soon that only w factors into the running time. Assuming that foresight, we will measure the problem size in w, so we write a recurrence for

$$T(w) = \text{how long it takes to run } \texttt{pow}(b, w) \text{ for any } b.$$

The function $T(w)$ is a recursive function that takes w, our problem size, and returns the number of steps (i.e., how long it takes) to run $\texttt{pow}(b, w)$ for any b.

To write $T(w)$, we will draw parallel to the code we wrote for `pow`. Once we examine the code, it is clear that there are two cases: Either we are in the small-instance case ($w = 0$) or we are in the recursive case (i.e., $w > 0$).

By inspection, clearly, when $w = 0$, we perform constant work (we test if w is 0, then we return 1), so $T(0)$ is $O(1)$. The case where $w > 0$ is more interesting: if $w > 0$, we carry out the following steps:

(i) we recursively call ourselves with $w - 1$; and

(ii) we multiply that result with b.

Therefore,

$$T(w) = \boxed{\text{cost of (i)}} \quad + \quad \boxed{\text{cost of (ii)}}$$

To determine the complexity of the recursive call, the crucial question to ask ourselves is, how large is the problem instance we are giving the recursive call? Answering this question in our case is easy: we are measuring the size in w and we are calling `pow` with $w - 1$. Hence, the cost of (i) is $T(w-1)$. And the cost of (ii) is constant, so we have

$$\begin{aligned} T(w) &= \underbrace{\text{cost of (i)}}_{=T(w-1)} \quad + \quad \underbrace{\text{cost of (ii)}}_{=O(1)} \\ &= T(w-1) + O(1), \end{aligned}$$

which, as our list of common recurrences below indicates, solves to $T(w) \in O(w)$.

Powering: Improvements via More Fancy Recursion

It is possible to write a more efficient function that achieves the same outcome. We will see how this is developed in a later chapter. For now, consider the following code:

```
long powHalve(long b, long w) {
    if (w==0) return 1;
    else {
        long t=powHalve(b, w/2);
        if (w%2==0) return t*t;
        else return t*t*b;
    }
}
```

Intuitively, it should be far better than the naïve version previously discussed. But can we analyze the time complexity of `powHalve`? To do this, we resort to recurrence relations.

Following the same argument as before, we have $T(0) = O(1)$. And for $w > 0$, we perform the following steps: (i) we compute $t = \texttt{powHalve}(b, w/2)$, (ii) depending on the parity of w, we either perform 1 or 2 multiplications. By now, we know that to compute t, we are calling `powHalve` with problem size $\approx w/2$, so the cost of (i) is $T(w/2)$. Furthermore, the cost of (ii) is constant. We conclude that $T(w) = T(w/2) + O(1)$, which we know from our list of common recurrences, solves to $T(w) \in O(\log w)$.

Example: Computing the Sum of Numbers

The second example deals with adding together numbers in an array. This is what we might write, recursively:

```
int sum(int[] a) {
    if (a.length==0) return 0;
    else if (a.length==1) return a[0];
    else {
        m = a.length/2;
        return sum(Arrays.copyOfRange(a,0,m)) +
            sum(Arrays.copyOfRange(a,m,a.length));
    }
}
```

Abstractly, in the code above, the following steps are performed:

1. Derive a copy of a with only elements in range 0 and m (exclusive).
2. Run sum recursively on this new array.
3. Derive a copy of a with only elements in range $m+1$ and the end of the list.
4. Run sum recursively on this new array.

Analysis. What is the running time? To find out, we will write a recurrence. Let $T(n)$ be the time to run sum on an array of length n. Immediately, we have $T(0) = O(1)$. And for $n > 0$, we know that there are two sources of running time: one due to recursive calls and the other due to basic operations, such as arithmetic and array slicing. The former is obvious: $2T(n/2)$ because we make two calls to sum, each on a problem of size $n/2$. The latter, as was shown before, is $O(n)$ because $m \approx n/2$ so each of `Arrays.copyOfRange(a,0,m)` and `copyOfRange(a,m+1,a.length)` is $O(m) = O(n)$. Hence, we have

$$T(n) = 2T(n/2) + O(n),$$

which solves to $T(n) \in O(n \log n)$.

Common Recurrences

We list a couple common recurrences, assuming $T(0)$ and $T(1)$ are constant:

- $T(n) = T(n/2) + O(1)$ solves to $O(\log n)$.
- $T(n) = T(n/2) + O(n)$ solves to $O(n)$.
- $T(n) = 2T(n/2) + O(1)$ solves to $O(n)$.
- $T(n) = 2T(n/2) + O(\log n)$ solves to $O(n)$.
- $T(n) = 2T(n/2) + O(n)$ solves to $O(n \log n)$.
- $T(n) = 2T(n/2) + O(n^2)$ solves to $O(n^2)$.
- $T(n) = T(n-1) + O(1)$ solves to $O(n)$.
- $T(n) = T(n-1) + O(n)$ solves to $O(n^2)$.

Exercises

Exercise 6.1. Show, using either definition, that $f(n) = n$ is $O(n \log n)$.

Exercise 6.2. Show, using either definition, that $g(n) = \log^2 n$ is $O(n)$. Throughout, we write $\log^k n$ to mean $(\log n)^k$.

Exercise 6.3. Show that $h(n) = 16n^2 + 11n^4 + 0.1n^5$ is *not* $O(n^4)$.

Exercise 6.4. Suppose a function `void fnE(int i, int num)` runs in $1000 \cdot i$ steps, regardless of what `num` is. Consider the following code snippet:

```
void fnA(int S[]) {
    int n = S.length;
    for (int i=0;i<n;i++) {
        fnE(i, S[i]);
    }
}
```

What's the running time in Big-O of `fnA` as a function of n, which is the length of the array `S`. Assume that it takes constant time to determine the length of an array.

Exercise 6.5. For each of the following functions, determine the running time in terms of Θ in the variable n.

```
void programA(int n) {
    long prod = 1;
    for (int c=n;c>0;c=c/2)
        prod = prod * c;
}

void programB(int n) {
    long prod = 1;
    for (int c=1;c<n;c=c*3)
        prod = prod * c;
}
```

Exercise 6.6. What is the running time of the following code?

```
int[] fooBar(int[] numbers) {
     int[] a = Arrays.copyOf(numbers);
     for (int j=1;j<a.length;j++) {
         int key = a[j];
         int i = j - 1;
         while (i>=0 && a[i] > key) {
             a[i+1] = a[i]
             i--;;
         }
     }
```

```
        return a;
    }
```

Determine both the worst-case running time and the best-case running time. The worst-case running on input of size n is the running time of the hardest input of that size—i.e., one that takes the longest to run. The best-case running time on input of size n is the running time of the easiest input of that size.

Exercise 6.7. Prove the additive and multiplicative properties of Big-O (Lemma 6.2) using either definition.

Exercise 6.8. Prove the additive and multiplicative properties of Θ (Lemma 6.4) using either definition.

Exercise 6.9. The brute-force algorithm for maximum contiguous subsequence sum (mccs) in Code 6.1 runs in $O(n^3)$ time. If we look closely, there is a nontrivial amount of redundancy. You will improve the running time to $O(n^2)$ time. (*Hint:* Consider how `thisSum` changes as `stop` increases.)

Exercise 6.10. Determine the running time recurrence of the following recursive function. The correct recurrence does not appear in our list of common recurrences.

```
int primSum(int[] xs) {
    if (xs.length == 1) return xs[0];
    if (xs.length == 2) return xs[0] + xs[1];
    else {
        int[] ys = Arrays.copyOfRange(xs, 2, xs.length);
        return xs[0]+xs[1]+primSum(ys);
    }
}
```

Exercise 6.11. Consider an alternative algorithm for summing up values in an array. Let X be an input sequence consisting of n floating-point numbers. To make life easy, assume n is a power of two—that is, $n = 2^k$ for some nonnegative integer k. To sum up these numbers, we use the following process, expressed in a Python-like language:

```
def hsum(X):  # assume len(X) is a power of two
  while len(X) > 1:
   (1) allocate Y as an array of length len(X)/2
   (2) fill in Y so Y[i] = X[2i] + X[2i+1] for i = 0, 1, ..., len(X)/2-1
   (3) X = Y
  return X[0]
```

Part I: Observe that the amount of work done in Steps (1)–(3) is a function of the length of X in that iteration. If $z = $ `X.length` at the start of an iteration, how much time is taken in that iteration as a function z (e.g., $10z^5 + z\log z$ or $k_1z^2 + k_2z$ for some $k_1, k_2 \in \mathbb{R}_+$)? Do not use Big-O; answer in terms of k_1 and k_2. To do this, we will make some assumptions here. For some $k_1, k_2 \in \mathbb{R}_+$:

- Allocating an array of length z takes time $k_1 \cdot z$.
- Arithmetic operations, as well as reading a value from an array and writing a value to an array, can be done in k_2 time per operation.

The answer for this part should look like the following:

$$\left(\quad \cdot \quad \right) z + \left(\quad \cdot \quad \right)$$

Part II: Then, we will analyze the running time of the algorithm (remember to explain how you get the running time you claim). To help you get started, make a table of how `X.length` changes over time if we start with X of length, say, 64. How does this work in general? *(Hint: The geometric sum formula will come in handy.)*

Exercise 6.12. In this problem, we will determine the worst-case and best-case behaviors of the following functions. The *best-case* behavior is the running time on the input that yields the fastest running time. The *worst-case* behavior is the running time on the input that yields the slowest running time.

(1) Determine the *best-case* running time and the *worst-case* running time of `method1` in terms of Θ.

```
void method1(int[] array) {
    int n = array.length;
    for (int index=0;index<n-1;index++) {
        int marker = helperMethod1(array, index, n - 1);
        swap(array, marker, index);
    }
}
void swap(int[] array, int i, int j) {
    int temp=array[i];
    array[i]=array[j];
    array[j]=temp;
}
int helperMethod1(int[] array, int first, int last) {
    int max = array[first];
    int indexOfMax = first;

    for (int i=last;i>first;i--) {
        if (array[i] > max) {
            max = array[i];
            indexOfMax = i;
        }
    }
    return indexOfMax;
}
```

(2) Determine the *best-case* running time and the *worst-case* running time of `method2` in terms of Θ.

```
boolean method2(int[] array, int key) {
    int n = array.length;
```

```
        for (int index=0;index<n;index++) {
            if (array[index] == key) return true;
        }
        return false;
    }
```

(3) Determine the *best-case* running time and the *worst-case* running time of `method3` in terms of Θ.

```
    double method3(int[] array) {
        int n = array.length;
        double sum = 0;

        for (int pass=100; pass >= 4; pass--) {
            for (int index=0;index < 2*n;index++) {
                for (int count=4*n;count>0;count/=2)
                    sum += 1.0*array[index/2]/count;
            }
        }
        return sum;
    }
```

Exercise 6.13. For each of the following Java functions:

(1) Describe how you will measure the problem size in terms the input parameters. For example, the input size is measured by the variable n, or the input size is measured by the length of array a.

(2) Write a recurrence relation representing its running time. Show how you obtain the recurrence.

(3) Indicate what your recurrence solves to (by looking up the recurrence in our list).

```
// assume xs.length is a power of 2
int halvingSum(int[] xs) {
    if (xs.length == 1) return xs[0];
    int[] ys = new int[xs.length/2];
    for (int i=0;i<ys.length;i++)
        ys[i] = xs[2*i]+xs[2*i+1];
    return halvingSum(ys);
}

int anotherSum(int[] xs) {
     if (xs.length == 1) return xs[0];
     else {
         int[] ys = Arrays.copyOfRange(xs, 1, xs.length);
         return xs[0]+anotherSum(ys);
     }
}
```

Exercise 6.14. Analyze the running time of the following prefix-sum implementation:

```
int[] prefixSum(int[] xs) {
    if (xs.length == 1)
        return xs;
    int n = xs.length;
    int[] left = Arrays.copyOfRange(xs, 0, n/2);
    left = prefixSum(left);
    int[] right = Arrays.copyOfRange(xs, n/2, n);
    right = prefixSum(right);

    int[] ps = new int[xs.length];
    int halfSum = left[left.length-1];
    for (int i=0;i<n/2;i++) ps[i] = left[i];
    for (int i=n/2;i<n;i++) ps[i] = right[i-n/2] + halfSum;

    return ps;
}
```

Chapter Notes

From early on, Aho, Hopcroft, and Ullman [AHU74] advocated the use of the asymptotic analysis of algorithms as a means to compare performance of algorithms. Both in his writing and elsewhere, Donald Knuth has been instrumental in popularizing use of asympotic notations in the study of algorithms. The book by Lehman et al. [LLM15] discusses mathematical techniques commonly used to analyze an algorithm's performance, including summation and recurrence techniques. For in-depth discussion of algorithms analysis, readers are encouraged to check out Cormen et al. [Cor+09]'s *Introduction to Algorithms*.